

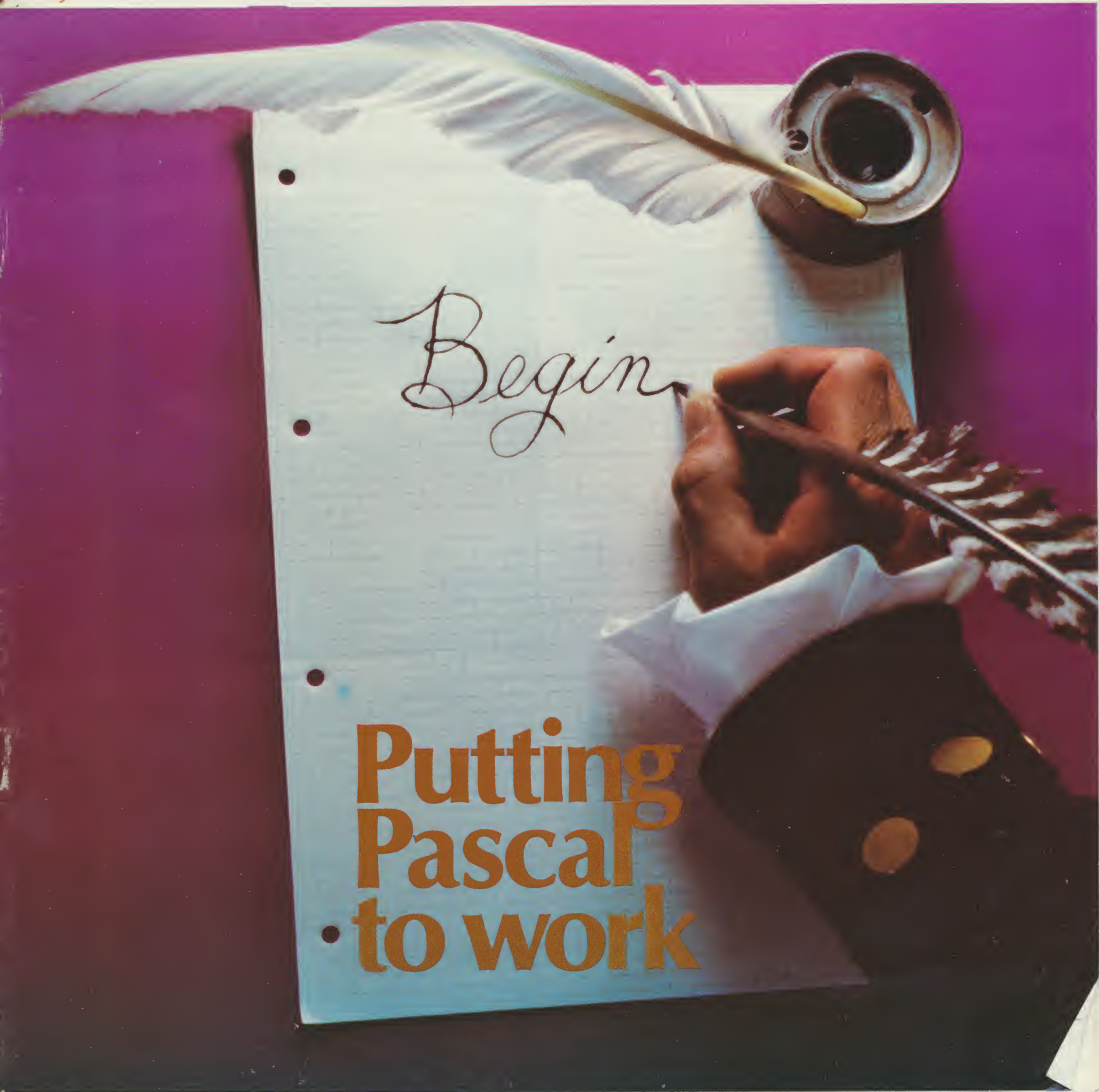
17 80
JUNE 7, 1979



A MCGRAW-HILL PUBLICATION

Electronics®

Reprinted from **ELECTRONICS**, June 7, 1979, copyright 1979 by McGraw-Hill, Inc., with all rights reserved.



Putting Pascal to work, Part 1

Language extensions, utilities boost Pascal's performance

More versatile data structures, tighter statement definitions, and system software promote program reliability

by Roger R. Bate and Douglas S. Johnson, *Texas Instruments Inc., Dallas*

Before Pascal emerged from academia, Texas Instruments Inc. was going over the language with a fine-tooth comb. Niklaus Wirth's brainchild was chosen for this scrutiny partly because it is easily understood and maintained—and partly because of the paucity of alternative choices.

Once Pascal was singled out, a working group proceeded to experiment with it in an attempt to extend its applicability across the entire spectrum of the TI computer product line. Efforts were also made to increase its usefulness in environments where a large number of users seek solutions to varied, complex problems.

This two-part article traces Pascal's three-year evolution at Texas Instruments. Part 1 focuses on TI's first crack at massaging the language to meet its corporate goal. The result, TI Pascal, retains the flavor of the original version, but has features that aid in program design, make it more successful in multiprocessing applications, and keep an even closer watch on sloppy programmers.

Part 2 is concerned with the most recent product of the continuing evolution of Pascal at TI: Microprocessor Pascal. This streamlined version of TI Pascal has some features removed and others added for users of the 16-bit 9900 family. A number of software tools accompany the firm's announcement of Microprocessor Pascal on June 1—a compiler, two executive programs, a text editor, a debugger, and a code generator.

-John G. Posa



□ There is no longer any question that Pascal is an important language for the future of programming small and large systems alike. If the academic community gave it a warm welcome, industry's more deliberate reaction is turning out to be even more enthusiastic. Adoption by industry, not surprisingly, has dictated various modifications to the standard Pascal as codified by Niklaus Wirth over a decade ago.

Not long ago, Texas Instruments Inc. completed an extensive study of existing high-level languages in order to settle on a standard corporate systems language (see "The history of Pascal at TI," p. 112). Pascal emerged as the language of choice; a program was then undertaken to enhance the language's abilities.

Part 1 of this two-part article looks at Pascal's advan-

tages for system programming and the facilities TI has added to it, including executive and utility programs introduced to aid in the software development process. Part 2, beginning on page 117, focuses on TI's Microprocessor Pascal and its two executive program packages.

A preliminary version of Pascal was drafted in 1968 by Wirth and associates, leading to a first compiler in 1970. After some evaluation and evidence of growing interest in the language, a revision was published in 1973 and a user's manual, describing what is now known as standard Pascal, in 1974. The original purpose of Pascal was to provide a language suitable for the teaching of programming as a systematic discipline. It was also hoped that reliable and efficient implementation would be possible on existing and future computers. It is based

The history of Pascal at TI

In 1976, Texas Instruments conducted an extensive internal study of existing high-order languages with the intent of adopting a standard system programming language. A series of selection criteria were adopted by a study committee to pit the spectrum of TI programming against the characteristics of existing languages. The most significant considerations were:

- **Maturity**—the language should have experienced several years of use to establish its limitations, range of applicability, user acceptance, and training ease.
- **Reliability**—the language should support testability, error detection, and error prevention. The most significant features that enhance reliability are strong typing of data, structured code, and structured data.
- **Efficiency**—its compilers and generated code should waste neither time nor space, for the sake of implementation on minicomputers and microprocessors.

Twenty languages were examined during this study. Fifteen of them were dropped fairly quickly when it became apparent that they were a long way from meeting the requirements or had compilers too complicated to implement on small machines. The features of five languages were examined in depth, and the final evaluation phase narrowed the field to two: Pascal and C.

Both are modern languages, with relatively little past use in comparison with languages like Fortran and Cobol, but both are being widely accepted in educational institutions and in some system programming projects. Pascal and C were carefully compared with the requirements in mind and, although they are quite different in their approaches, they turned out to be fairly equally matched as system programming languages.

However, TI had considerable experience with a Pascal-based language called Process Design Language II (PDL-2), which was designed for the Ballistic Missile Advanced Technology Center as a language for ballistic missile defense real-time software. PDL-2 extended standard Pascal, adding vector operations and capabilities for multiprocessing and synchronization. The language was implemented on the TI Advanced Scientific Computer and later transported to the CDC 7600. The very favorable experience with this language in reducing the cost of software development and in improving the readability of complex real-time programs resulted in the decision to adopt Pascal as the standard language for TI.

A steering committee met on an irregular basis to oversee the work of a full-time working group that defined a language based on Pascal but had adequate extensions for multiprocessing in a real-time environment. After agreement was reached on the definition of the language, a compiler was written to run on the TI 990 and the IBM 370. The TI Pascal compiler was made commercially available in 1978 by the TI Digital Systems Group as part of the standard software for the TI 990 minicomputer.

After the release of the TI Pascal compiler, it became clear that the language would be extremely useful for programming microprocessors for industrial and control applications. For that reason, a variant called Microprocessor Pascal has been designed. The language has fewer extensions than TI Pascal and is therefore somewhat more easily implemented on small computers. The Microprocessor Pascal compiler, in fact, will run on an FS 990 floppy-disk system that uses a TI 9900 microprocessor as its central processing unit.

on the language Algol 60, from which most of its control mechanisms are drawn. But it includes a number of innovations in the way data structures are defined. Algol 60 is not a subset of Pascal, although in much of its regularity and style the younger language is very similar to the older one.

Simple and structured statements

The actions described by a Pascal program are contained in the language's underlying statements. These statements are of several kinds. Simple statements, such as the assignment statement, allow the value of an expression to be assigned to the value of a variable. Procedure statements cause code located outside the currently executing sequence to be invoked. And goto statements force program control to jump to a labeled statement located elsewhere. (In hand-written Pascal programs and examples, word delimiters—reserved words—are generally underlined.)

Structured statements provide a degree of control over which of the simple statements are executed. The conditional if and case statements select a group of statements based upon the value of an expression that is computed prior to the selection. Repetitive statements such as while, repeat, and for permit repetitive evaluation of a statement or group of statements, depending upon the value of an expression. This value is altered during the execution of the controlled statements, thus allowing the

statements to be repeated a desired number of times. In these respects, Pascal differs little from Algol and other Algol-like predecessors.

It is Pascal's data-structuring facilities that set the language apart from its forerunners and make it attractive for serious programming. Pascal greatly expands the concept of the data type. Although some of the primitive types are also in Algol 60, Pascal allows the construction of more complex types from the primitives. Furthermore, the language allows the user to define new data types and to name these with his own identifiers.

Each variable must be declared to have a type. When variables are used, the compiler checks to make sure that they are used consistently. This strong type checking, as it is called, pervades the language. It increases program reliability because the user is forced to define the intended use of each variable and because the aid of the compiler is enlisted to make an exhaustive check of his program to assure compliance with his stated intent. In this way, the benefits of an extremely flexible data-definition facility are provided along with automatic checking to insure correct use of complicated structures.

The simple data types primitive to Pascal include the scalar type and the subrange type. The scalar types defined in standard Pascal are integer, with values in the subset of whole numbers defined by the implementation; real, a subset of the real numbers; Boolean, with the values true and false; and char, the set of characters

TABLE 1: THREE OF PASCAL'S STRUCTURED DATA TYPES

	Arrays	Records	Sets
Definition	A: array [0..99] of real; B: array [color] of integer; C: array [Boolean] of array [color] of char;	R: record name: packed array [1..10] of char; age: 0..120; married: boolean; eyes: color end;	type S = set of 0..15; days = set of (mon, tue, wed, thu, fri, sat, sun); prim = set of color; var week, work : days;
Access	A [13] := 3.14159; B [green] := -1; C [true, blue] := 'B';	R .name := 'Pocahontas'; R .age := 16; R .married := false; R .eyes := green;	work := mon..fri; week := work + sat, sun;

recognized, again, by the particular implementation.

Additional scalar types can be defined by the user as ordered sets of elements, each assigned an identifier and consisting of an enumeration of other identifiers, thus:

color = (red, orange, yellow, green, blue);

error = (overflow, index, parity, protect);

Subrange types are user-defined subranges of other scalar types, such as:

percent = 0..100;

byte = 0..255;

Easy access

Pascal's complex or structured data types include arrays, records, sets, and files. An array is a data structure consisting of a fixed number of components, all of the same simple type. Access to an element of an array is provided by giving an expression for an index. This expression is evaluated to give a value of the proper type; the value is then converted into an integer that is the index of the desired array element. The resulting element value can be used in further operations consistent with its type.

A record consists of elements with a fixed number of components, defined by the programmer. In records, however, the components may be of different types. Each component or field is accessed by a name, called the field identifier. Fields of records are accessed by giving the record-variable identifier and the field identifier. The resulting value has the type specified for that field in the record definition.

The set type is a structure that represents a collection of objects of a given base, where the base is an enumerated simple type. For instance, the base may be a subrange of integers or a list of identifiers. A variable that is declared to be of the set type takes on values that are sets of elements of the base type.

One representation of the set type is a bit string of length equal to the number of elements in the base type, each bit being 0 or 1 depending upon whether the corresponding element is a member of the set or not. Set types are manipulated by operations that test set membership and return a Boolean value. Other set operators that apply are union, difference, intersection, and assignment. Table 1 gives examples of the array, record, and set types, as well as examples of references to the

variable elements of the arrays, fields of the records, and the variables of the sets.

A file comprises a sequence of elements, all of the same simple type. It is accessed from one end and elements are added on the other end. Thus the number of elements in a file can change dynamically. A typical use of files is for input and output.

File elements are accessed by a read statement that names the file variable and the variable into which data is to be entered. Similarly, data values are added to files by a write statement that names the file variable and gives an expression to be evaluated. The compiler checks to make sure that the variable and the expression are of the same type as the components of the file.

Finally, the pointer type and the capabilities it provides are worthy of special mention. Declaration of a pointer yields a type which is associated with some other variable, such as a record. The statement new (pointer variable) creates a new instance of the type pointed to and sets the pointer variable to the address where the newly created data object is located. Data structures such as lists, trees, and graphs can thus be defined by linking their elements together with pointers. Each element in one of these structures may have one or more pointers to other elements in the same structure.

There are several major advantages to these data-defining capabilities of Pascal. First, the user may describe his data structures precisely to a compiler in order that his use of variables may be checked for consistency with their definitions. Second, the user may define very complex data structures to suit his needs without concerning himself with the actual implementation of the structure. He simply accesses it by indexing or by naming the part he wants. He does not, for example, have to remember (as he would in Fortran) which elements of an array represent "age" and which represent "social security number" since he can define the fields of a record with appropriate names and types.

A third advantage of data-structure definitions is that they may be changed to add new fields or to rearrange fields in records without a corresponding change in the source code where these elements are accessed. The consequences of changes in data definitions are automatically propagated by the compiler into object code. These capabilities are not only convenient, they are important


```

PROCEDURE SCAN
  (VAR A: ARRAY [1..?] of CHAR; VAR I, L: INTEGER);

  {This procedure scans for the next identifier in the string A.
   I is the current index in the string, and L is the length of
   the identifier found. If next non blank in A is not a letter,
   a length of zero is returned. An identifier is a letter
   followed by any sequence of letters and digits.}

  Begin {SCAN}
    While I < UB(A) and A[I] <> ' ' do           " skip blanks
      I := I + 1;
    L := I;                                       " save initial index
    If A[I] in ['A'..'Z'] " if first char is letter, continue
      then
        begin
          LOOP: While I < UB(A) do " scan letters and digits
            Case A[I] OF
              'A'..'Z': I := I + 1;
              '0'..'9': I := I + 1;
              otherwise escape LOOP;
            end {Case};
            L := I - L + 1; " length of scanned identifier
          end
        else
          L := 0; " if first char not letter, return zero
        end {SCAN};

```

in increasing the reliability of programming efforts and reducing their cost.

In Pascal the primitive types all have imposed on them the usual arithmetic and logical operations seen in other languages—for instance, assignment, addition, intersection of sets, and disjunction of Boolean expressions.

Extensions

While standard Pascal has most of the features needed for system implementation in general, some characteristics of the language make it awkward in environments where a large number of programs are written by many people. This is particularly true in an atmosphere where libraries of general utility programs must be provided. For this reason, a number of extensions and modifications to the language are included in TI Pascal to make it more suitable for industrial use.

Standard Pascal defines only sequential fields that are read from one end and to which elements can be added on the other end. In many data-base applications, however, it is desirable to access elements of a file by index. Such random access is equivalent to indexing into an array, but the array cannot be dynamically extended. This desirable combination of facilities is implemented by giving an additional parameter to the read and write statements: an expression, evaluating to an integer, that describes a position in the file.

Standard Pascal does not provide statically allocated

variables in the way Fortran's common and Algol's own declarations do. Memory space for statically allocated data is computed and fixed when the program is compiled. In Algol or Pascal, on the other hand, the underlying machine model is an idealized stack-based computer. The local variables of a routine appear while it is executing and they disappear when the routine is exited. The stack used for the variables collapses.

In other words, Pascal variables are restricted in scope. The scope of any item defined within a routine or procedure is that portion of the program in which its definition is recognized.

So standard Pascal cannot extend the definition of a variable in one routine into another routine in the way Fortran's common statement and Algol's own statement do. Many programmers feel this is essential, so Fortran's common statement is implemented in TI Pascal. This permits external (Pascal, Fortran, or Cobol) subroutines called from a Pascal program to share variables and also yields the equivalent of Algol's own variables. Control over access to these variables is provided by requiring that such access be in the scope of the appropriate common declaration and that it be specifically allowed by an access declaration in the calling program module.

The restriction to fixed-length arrays and sets in Wirth/Jensen Pascal makes it extremely difficult to write general utility routines that are compiled separately from the user program, such as sorts and searches. For

TABLE 3: THE SCOPE OF NESTED PASCAL PROCEDURES

Scope of:	A	B	C	D	E
Procedure A; variables for A;	• •				
Procedure B; variables for B;	• •	• •			
Procedure C; variables for C; body of C;	• • •	• • •	• • •		
Procedure D; variables for D; body of D; body of B;	• • • •	• • • •		• • •	
Procedure E; variables for E; body of E; body of A;	• • • •				• • •

in minor ways. Nevertheless, the resulting language seems to be much more capable of being precisely defined.

For instance, the control variable in a standard Pascal for statement is presumably undefined outside the for statement, yet no restriction on using the control variable is enforced by standard Pascal. A restriction was added to the TI compiler that confines the control variable's use to its for statement.

Another example concerns side effects, or alterations in a function's operational environment caused by the function itself. Side effects are generally considered undesirable, but they are not all eliminated by the standard Pascal compiler. A side effect is an action that is extraneous to basic execution of a function; this action may or may not be defined in the language. For example, if a square root routine is passed an argument *x*, it should not change the value of *x* in the process of returning a value for the square root of the number.

TI Pascal does not allow any function to change the value of a parameter it is passed. Nor does it allow any function to alter the value of global variables or to call other procedures.

The capability of Pascal's case statement was enhanced by providing an otherwise clause. The otherwise clause is a default alternative, selected if the case index doesn't match any of the case labels. Subrange case labels were also implemented: these allow the programmer to specify a range of values for a case alternative. If the same alternative is to be indicated by values 1, 2, and 3, standard Pascal requires a label for each value. TI Pascal allows a single label for the three, using the expression 1. . 3. These case statement extensions are illustrated in Program 1. It also illustrates the use of a dynamic array and an escape label.

One of TI's more controversial changes is a return to the precedence for Boolean operators used in Algol and Fortran. Wirth Pascal's precedence for Boolean operators was unlike that of either Algol or PL/1—it defined a whole new choice in this important area.

This tightening up of the language may inconvenience

some programmers, but program maintenance and reliability is substantially improved. And the resulting code is more readable: it contains few if any hidden surprises. Table 2 summarizes some of the modifications and extensions in TI Pascal.

As might be supposed from the number of extensions in TI Pascal, the compiler is somewhat larger than the standard compilers and does run more slowly. Nevertheless, the optimization that has been included provides for reasonably compact and efficiently executed code.

The TI Pascal compiler has three phases, which may be roughly described as a lexical-analysis phase, a translation into an intermediate language, and code generation, in which the intermediate language is translated into the code of the target machine. This structure was chosen in order to maximize transportability of the compiler; that is, to make it possible for programs written in the language to be carried from one machine to another. This portability has been extremely successful. TI Pascal compilers and other Pascal programs have been transported between IBM's 370 and TI's 990 and 980 minicomputers without significant problems. An added benefit has been the simplicity of designing cross-compilers.

The environment

The introduction of a new language requires more than the writing of a compiler. There is a set of associated software tools that must go along with any compiler to help the writer use the language efficiently and to increase his reliability and productivity. Several of these tools are included in the package distributed with the compiler, because they are essential or highly desirable for Pascal programming.

The most significant tool is a configuration processor, depicted in Fig. 1. This automates most of the clerical work associated with the coding of a very large program, which may consist of many modules. The user defines a hierarchy of modules in libraries to match the hierarchy of declarations in the Pascal program. (Table 3 shows the scope of routines and their variables in a typical program.) The configuration processor is then able to select, in proper order, the declarations of global variables and routines that must be included for successful compilation of one or more modules of a program and to retain control over the object modules that are produced.

These declarations make all the necessary type definitions and variables available when the module body is being compiled. In this way, the minimum amount of information is handled for each compilation. In addition, separate compilation in a scoped language becomes a very feasible and reliable activity. This clerical tool has been found to be essential for large Pascal programs.

Also provided is a program that accepts Pascal source-program text and prints it with standard indentations to display its structure. This is useful for documentation, to show the programmer the structure that he has actually created, and as a debugging tool. It is also possible to print out an assembly language representation of the machine code generated by the TI Pascal compiler, if it is required for specific debugging activity, with a reverse assembler program.

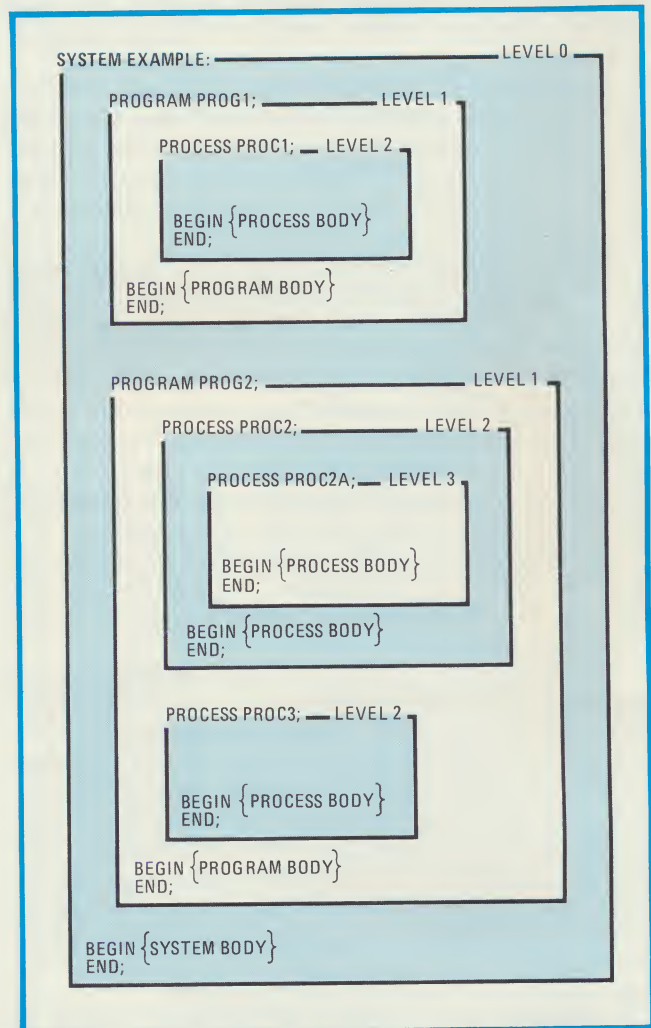
Pascal software supports real-time multiprogramming on small systems

by Roger R. Bate and Douglas S. Johnson,
Texas Instruments Inc., Dallas

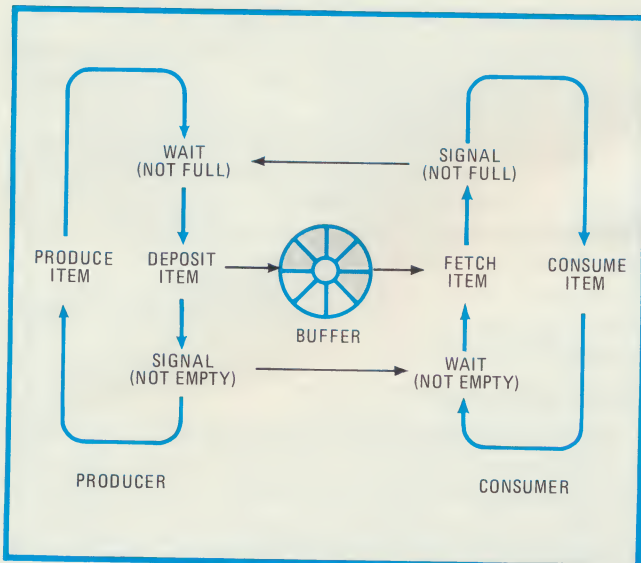
□ Industrial and other real-time control situations call for special software support to implement a high-level language on a small system. If the system does not include a general-purpose operating system, some sort of multitasking executive is needed if a language like Pascal is to be exploited.

Texas Instruments Inc. has developed support of this type for a subset of TI Pascal (see Part 1 of this two-part article) called Microprocessor Pascal. A superset of standard Pascal, Microprocessor Pascal has features that allow multiple sites of execution (called processes), synchronization among processes via semaphores, and access to the communication-register unit of TI's 9900 family of microprocessors.

A user of Microprocessor Pascal may develop software for general-purpose applications using one of two host computer systems: the single-user FS 990 floppy-disk-based minicomputer, or the multi-user DS 990 hard-disk-based minicomputer. The Microprocessor Pascal system comprises a variety of software support tools, including an intelligent interactive editor for



1. Nesting. There are two special types of Microprocessor Pascal processes: programs and systems. A system is a process in which execution begins; it initializes global variables and starts up its constituent programs. A program may contain nested processes, and multiple programs may execute concurrently within a system.



2. Semaphores. A semaphore represents some event upon which processes synchronize. These exist as a variable with a counter and a queue of suspended (waiting) processes. Semaphores provide flexible, efficient, user-controlled synchronization.

source preparation, a compiler that generates interpretive code, a code generator that supplies native object code, and an interactive debugging interpreter.

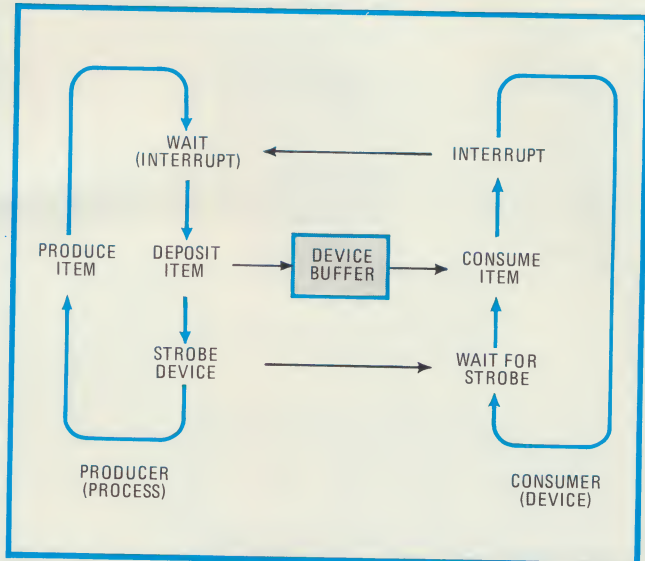
There are also two powerful executive programs which support the execution of the user system on the target processor. These memory-resident executives provide for multiprogramming, interrupt handling, interprocess communication through input/output files, dynamic creation and reclamation of processes, and multiple-priority process scheduling.

Executive benefits

A major benefit of Microprocessor Pascal and either of its executive kernels is that the system designer need not specify system-wide standards and conventions for intermodule parameter passing, register allocation, or task management. These design decisions are automatically made in the way the Microprocessor Pascal system is mapped onto the architecture of the 9900 family.

The input/output facilities of the system provide a standard high-level intermodule interface that permits the development of modular software libraries from which users can select exactly those components needed for a particular application. The executive kernels, too, emphasize software modularity; their components are provided in a form that permits the user to include in his applications only those features that are essential.

The executive components are provided in two versions that correspond to the two types of output from the compiler: interpretive and native object code. The Microprocessor Pascal interpretive executive is generally used for applications where the program-code compactness achieved with language interpretation is more important than the associated decrease in execution speed. The second Microprocessor Pascal executive supports the execution of native code and can therefore be used in time-critical applications. Both versions of the executive provide a user with identical capabilities.



3. Interrupts. A hardware interrupt is a stimulus from the outside world, generally used to pass a signal from a device to a process. Correspondence is established between an interrupt and a semaphore; the signal elicits response from a waiting process.

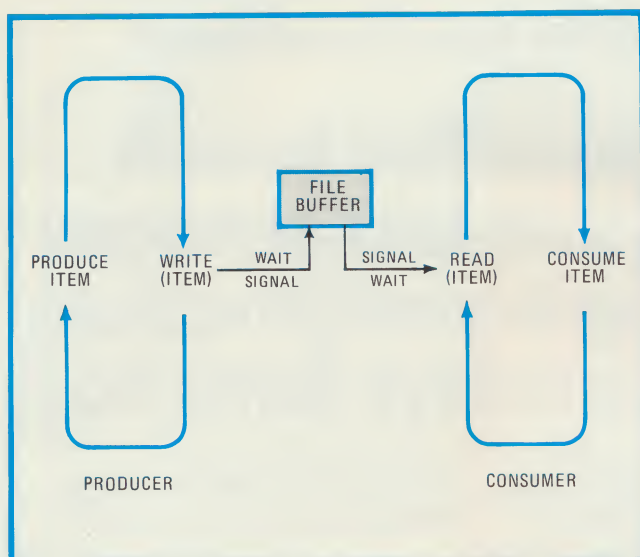
A conventional Pascal program consists of a main program along with zero or more subprograms, called procedures or functions. Execution of such a program is done serially, from one statement to the next; at any time, execution occurs at a single point in the program. However, the practice of having several sites of concurrent execution within one program is often desirable. To permit this multiprogramming, the concept of a process has been introduced into Microprocessor Pascal. A process is a separately executing entity having its own run-time environment for its data.

Within the framework of the Microprocessor Pascal system, the design of a multitasking application is to a great degree reduced to the assimilation of various processes, each of which is basically a sequential Pascal program. Well-defined techniques and executive components manage the time-dependent interactions among these processes. The user can therefore concentrate on the algorithms that comprise an application without concern for the construction of the executive under which they execute.

Multiprogramming

In a stand-alone situation, multiprogramming and interrupt handling are often desirable and in some cases essential. An industrial process-control problem, for example, could be solved by preparing a process to oversee each type of device in the system. In very complex systems, process allocation could be even finer: several processes could be used, one for each mode of operation of each device. With processes the user may also write programs to service interrupts and devices, and thus realize more general multiprogramming.

Microprocessor Pascal processes are synchronized so they can communicate with each other. They are scheduled (selected for execution) based on process readiness and process priority. Readiness indicates whether a process may proceed or if it must wait for some condition



4. Logical I/O. Interprocess communication is treated as logical input/output. A process may read data elements written asynchronously by another process. The executive performs wait and signal operations to facilitate the communication.

to be satisfied first. Priority indicates the relative urgency of the process. The lower its priority number, the more time-critical a process is.

There are two special types of Microprocessor Pascal processes: programs and systems. A program is a self-contained process. It is declared with the Pascal keyword program, and has no external data available to it except possibly through common variables. A program may contain nested processes, and one or more programs may execute concurrently within a system.

A system is the process in which execution begins. It bootstraps itself by initializing the global parameters it needs (parameters defined in all processes used in the system) and by starting up its constituent programs. A system may not have any variables of its own, except those defined through common declarations. Figure 1 shows programs and processes nested within a system.

Every program or process has associated with it two dynamic data structures to manage memory. One of these is called the stack, and the other is called the heap. The stack is an area allocated to the declared variables of the program or process and its procedures. The heap holds dynamically allocated data structures, which are not declared but are created and destroyed by the two procedures new and dispose. With these features, all processes (and the routines within them) are naturally re-entrant. A re-entrant routine's code does not change during execution. A process using re-entrant code can be stopped at any point and the program re-entered by a different user or process.

Microprocessor Pascal extensions are designed to aid the user in the following ways:

- Process declaration is distinct from the declaration of a Pascal procedure or function.
- Process declarations may be nested, and the Pascal scope rules of global variables are enforced as usual.
- Processes, like procedures, may declare parameters. The start statement allows the passing of process param-

eters with full type checking at compilation time.

- Variables within the scope of a process are guaranteed to exist even if the processes that are its higher-level ancestors have terminated.
- The program construct is a special case of a process that has no variables global to it. Its resources are given special treatment by the executive.
- Any process or program that is within scope can be concurrently executed with the start statement. To allow all program declarations (declared at level one) to be in scope, the system construct (at level zero) must contain all program declarations.

Interprocess communication

Real-time programming concerns the control of events in the physical world. The central problem in any real-time environment is that the computer must be able to receive and react to events as fast as they arrive, lest it fall behind the real world. The stimuli of externally generated events are passed to processes within the computer by means of hardware interrupts.

When an event occurs, the process may service it, proceed, or take some other action. Until the event happens, however, the waiting process is suspended and does not compete for a processor's time.

The event upon which a process waits may be generated by another process. For example, consider processes that have related responsibilities; ones that must communicate with each other, either through shared memory or some form of message-passing protocol. Successful communication from one process to another requires synchronization between the processes to ensure that they do not interfere with one another. If a sending process wants to place an item into the next available space in a buffer, for instance, it must ensure that the receiving process does not modify pointers to the buffer until the transfer is complete.

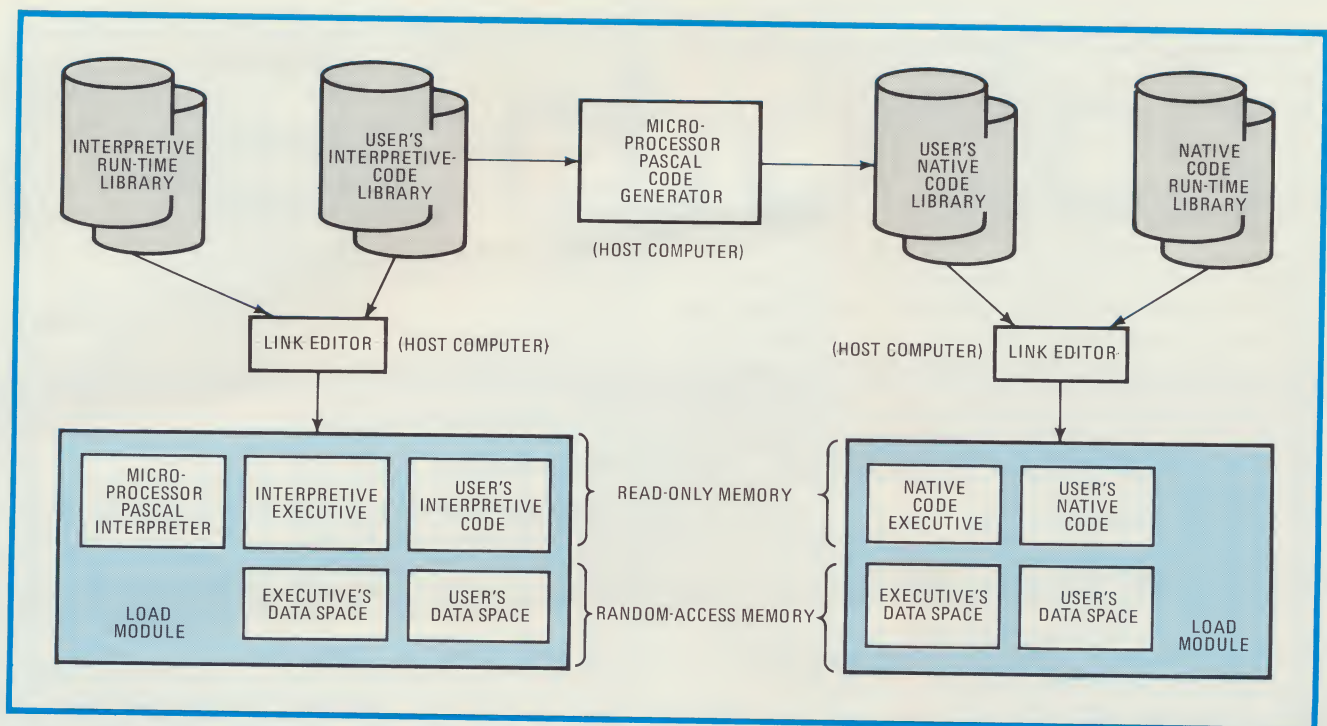
In general, a system designer cannot determine *a priori* the progress rates of the individual processes, and some low-level primitive structure must be provided for the synchronization of otherwise asynchronous processes.

Synchronization of process actions with other activities is achieved by means of an event that causes all actions to concur or agree in time. The Microprocessor Pascal executive provides two mechanisms for synchronizing external events: semaphores and interrupt handling. These mechanisms may be used to synchronize several concurrent processes on a single processor or on multiple processors if shared memory and interrupt circuits are provided.

Semaphores

The term semaphore stems from the semaphores of railroading that are used to synchronize access to sections of track. A semaphore represents some event upon which processes must synchronize, and is implemented in Microprocessor Pascal as a variable consisting of a counter and a (possibly empty) queue of suspended processes. The two basic operations performed on a semaphore are wait and signal.

A process may ensure that an event has occurred before proceeding by performing a wait operation on the



5. Two routes. For the interpretive environment, the user's code is joined with modules from the interpretive library and linked to a load module. Alternatively, the user's code can be run through a code generator, and the interpreter omitted from the load module.

associated semaphore. If the event has already occurred, the process continues execution; otherwise, the process is suspended by the executive until the event occurs.

A process may signal the occurrence of an event by performing a signal operation on the associated semaphore. If another process is waiting for the event, the executive makes that process ready for execution; otherwise the occurrence of the event is recorded by incrementing a counter in the semaphore until a subsequent wait operation occurs for that event. In either case, the process that performed the signal operation remains in the ready state.

Microprocessor Pascal implements semaphore counting in such a way that an occurrence of an event is not lost even if no process is waiting when an event occurs. The executive keeps a count in a semaphore of the number of events that have occurred (by signals), but have not been received (by wait operations).

Semaphores provide efficient user-controlled synchronization among processes and are flexible enough to be useful in the solution of complex application-dependent problems such as resource scheduling and interprocess communication. An example of a communicating pair of processes is given in Fig. 2. The relationship between synchronization and data flow through a shared bounded buffer is illustrated.

Interrupts

A hardware interrupt is a stimulus from a processor's external environment to pass an event to a process within the processor. Interrupt handling can be viewed in various ways that differ primarily in the type of interface used. The Microprocessor Pascal executive technique for interrupt handling uses the event mechanism of semaphores. A correspondence is established between an

interrupt and a semaphore such that when the interrupt occurs, the executive interrupt handler performs a signal operation on the semaphore, thus activating a waiting process, if one exists, to respond to the interrupt. Figure 3 illustrates the relationship between a producer (a process) and a consumer (a device such as a printer). The producer waits for an interrupt to show that the device is ready, then deposits data in a buffer. The device, running at its own speed, reacts to the strobe by consuming the data, then signals the producer with an interrupt. This device-handling technique is just one special case of the general semaphore mechanism shown in Fig. 2.

Logical files

Files are usually associated with storage media such as disk or magnetic tape. However, many operating systems also allow devices to be treated as files in a consistent manner. Therefore, the term logical file is used to indicate any communication medium with which processes can perform logical input/output; that is, I/O that is device-independent. With this definition, a logical file can be a disk file in a directory, a video display terminal, a keyboard, a card reader, a line printer, etc. Due to the uniform interface used in logical I/O, programs do not have to be aware of the unique characteristics of devices or storage media.

In Microprocessor Pascal, logical files are manipulated through variables of the `file` type, described in Part 1 of this article.

Allowing logical files to include interactive devices such as video display terminals (VDTs) permits the sequence of data elements to be generated in real time by an intelligent source, such as a person at a keyboard, rather than by simply reading previously-generated data

from a storage medium. The generation of this data may also be influenced by a program's output, which is produced in real time and displayed on the screen of the VDT. In essence, both the program and the person are influenced by each other in their real-time interactions, thus forming a system of two cooperating processes.

Figure 4 shows the Microprocessor Pascal approach to interprocess communication. It is treated as a form of logical I/O; that is, cooperating processes may communicate with each other using logical files. One process may read data elements that are being written concurrently by another process in conjunction with the executive-handling wait and signal operations. This is very similar to the interaction described above because the input is generated and the output consumed in real time by the processes rather than being stored on or retrieved from some storage device.

Logical file I/O provides a consistent interface between system components, which include both hardware devices and software programs. This allows systems to be constructed from modular components, each of which is understood in terms of its inputs and outputs. In this way, the interfaces between the components form a nearly complete definition of the system. Each component can be designed, implemented, and tested independently, isolated from other units to verify that it performs its required functions.

Software tools

The Microprocessor Pascal editor is designed to help with the creation and modification of program source files. It is interactive and uses a video display terminal. Changes may be made to the file simply by typing over the old text with new text or by adding or deleting complete lines. Commands that may be invoked from the editor cause a limited form of syntax checking of the Pascal statements that are in the current file. Detected errors are marked by the cursor and described so that the user can make an immediate change to correct the error. Syntax checking is fast enough so that it does not significantly delay the programmer. In addition, as the programmer creates his source text, the editor helps by providing the correct indentation of structured statements. Figure 5 shows how the editor is used to fashion load modules.

The Microprocessor Pascal compiler translates source code into interpretive code for an abstract stack computer. This code may be executed interpretively using the debugger or it may be used as input to the code generator. This interpretive code has several advantages. It consumes roughly half the memory required for 9900 machine code and it can be created very quickly by the compiler. Since it is executed interpretively, debugging aids such as breakpoints and symbolic lookups can be readily incorporated.

The Microprocessor Pascal code generator accepts the interpretive code from the compiler as input and converts it into 9900 object code. It attempts fairly sophisticated optimization and allocation of registers to variables, and generates efficient code for references to the address space of the communications-register unit through which peripheral devices are controlled. The generated code is

re-entrant and position-independent; that is, the code can be loaded at any address without modification.

The Microprocessor Pascal debugger supports symbolic references to module names, file names, and common names. Statements can be referenced by Pascal statement number. Either single processes or sets of processes can be debugged at one time. Breakpoints can be used to stop the execution at any point by specifying the Pascal statement number of a particular module. When execution is suspended, the status of execution can be examined and data can be modified at this point if desired. Traces of the current execution of the process scheduling level, module entry and exit level, or statement level are also available.

Standardization

Since Pascal was first released, many implementations have been developed at universities and in industry. Many of these versions differ by taking full advantage of the looseness of the standard Pascal definition, and some contain extensions. In an attempt to deal with this proliferation of versions of Pascal, the Pascal Users Group (PUG) has supported a working group with the International Standards Organization, which appears to be espousing little change from the original language.

A different point of view was represented by a workshop conducted in July 1978 by Dr. Kenneth Bowles at the University of California at San Diego. At this meeting, members of PUG and industry attempted to define a set of standard extensions. However, this workshop was unable to arrive at any sort of agreement.

Recently, the American National Standards Institute (ANSI) has organized the X3J9 committee on Pascal standards, and the Institute for Electrical and Electronic Engineers (IEEE) has formed a subcommittee that now cooperates with that effort [*Electronics*, May 24, p. 98].

These efforts should be supported by industry in order to achieve a language that will serve the needs of industry and at the same time provide a degree of standardization that will make efficient use of Pascal possible. The advantages of Pascal over other existing languages in common use merit a substantial effort.

Pascal was the basis for all four of the proposed languages accepted for contract by the Department of Defense for the development of its common high-order language, now called Ada [*Electronics*, May 24, p. 64]. While it is true that Ada departs significantly from Pascal and makes some unnecessary changes in the syntax, most of the good features of Pascal have been retained. Extensions that provide the capability for systems and real-time programming are likely to be included in the final product.

It appears that the DOD effort has the potential of producing the long-awaited successor to Fortran. If sufficient momentum is maintained by DOD and industry, this can provide a great boon for software development, in the form of a powerful modern language useful for the bulk of programming tasks. However, it will require considerable effort to get over the critical-use threshold such that it can be regularly taught in schools and considered a part of the equipment of programmers and software development facilities everywhere. □